# Chapter 3

# Test Generation from Finite State Models

*The purpose of this chapter is to introduce techniques for the generation of test data from finite state models of software designs. A fault model and three test generation techniques are presented. The test generation techniques presented are: the W-method, the Unique Input/Output method and the Wp method.*

## 3.1. Software design and testing

Development of most software systems includes a design phase. In this phase the requirements serve as the basis for a design of the application to be developed. The design itself can be represented using one or more notations. Finite state machines, state charts, and Petri nets are three design notations that are used in this chapter.

A simplified software development process is shown in Figure 3.1. Software analysis and design is a step that ideally follows requirements gathering. The end result of the design step is the *design* artifact that expresses a high level application architecture and interactions amongst low level application components. The design is usually expressed using a variety of notations such as those embodied in the UML design language. For example, UML state charts are used to represent the design of the real-time portion of the application and UML sequence diagrams are used to show the interactions between various application components.

The design is often subjected to test prior to its input to the coding step. Simulation tools are used to check if the state transitions depicted in the state charts do conform to the application requirements. Once the design has passed the test, it serves as an input to the coding step. Once the individual modules of the application have been coded, successfully tested, and debugged, they are integrated into an application and another test step is executed. This test step is known by various names such as *system test* and *design verification test*. In any case, the test cases against which the application is run can be derived from a variety of sources,

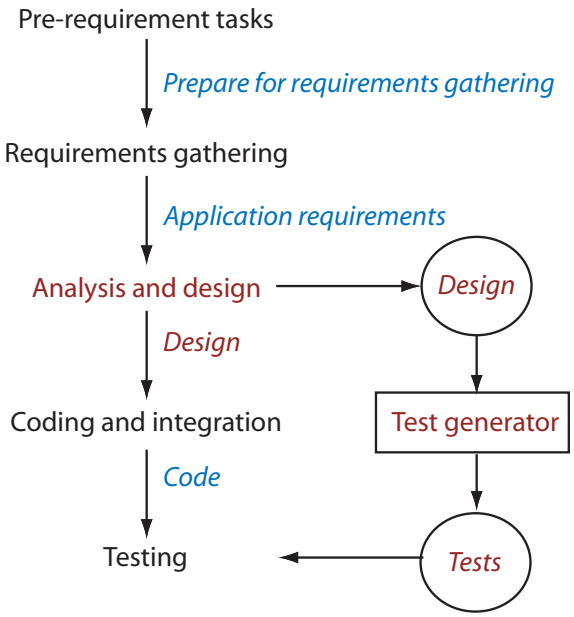


Figure 3.1: Design and test generation in a software development process. A *design* is an artifact generated in the Analysis and Design phase. This artifact becomes an input to the Test Generator algorithm that generates tests for input to the code during testing.

design being one of the sources.

In this chapter we will show how a design can serve as source of tests that are used to test the application itself. As shown in Figure 3.1, the design generated at the end of the analysis and design phase serves as an input to a *test generation* procedure. This test generation procedure generates a number of tests that serve as inputs to the code in the test phase. Note that though Figure 3.1 seems to imply that the test generation procedure is applied to the entire design, this is not necessarily true, tests can be generated using portions of the design.

We introduce several test generation procedures to derive tests from finite state machines, and state charts. The finite state machine offers a simple way to model state-based behavior of applications. The state chart is a rich extension of the finite state machine and needs to be handled differently when used as an input to a test generation algorithm. The Petri net is a useful formalism to express concurrency and timing and leads to yet another set of algorithms for generating tests. All test generation methods described in this chapter can be automated though only some have been integrated into commercial test tools.

There exist several algorithms that take a finite state machine and some attributes of its implementation as inputs to generate tests. Note that the finite state machine serves as a source for test generation and is not the item under test. It is the implementation of the finite state machine that is under test. Such an implementation is also known as *Implementation Under Test* and abbreviated as IUT. For example, an FSM may represent a model of a communication protocol while an IUT is its implementation. The tests generated by the algorithms we introduce in

this chapter are input to the IUT to determine if the IUT behavior conforms to the requirements.

In this chapter we shall introduce the following methods: the W-method, the transition tour method, the distinguishing sequence method, the unique input/output method, and the partial-W method. In Section 3.9 we shall compare the various methods introduced. Before we proceed to describe the test generation procedures, we introduce a fault model for FSMs, the characterization set of an FSM and a procedure to generate this set. The characterization set is useful in understanding and implementing the test generation methods introduced subsequently.

## 3.2. Finite state machines

Many devices used in daily life contain embedded computers. For example, an automobile has several embedded computers to perform various tasks, engine control being one example. Another example is a computer inside a toy for processing inputs and generating audible and visual responses. Such devices are also known as *embedded systems*. An embedded system can be as simple as a child's musical keyboard or as complex as the flight controller in an aircraft. In any case, an embedded system contains one or more computers for processing inputs.

An embedded computer often receives inputs from its environment and responds with appropriate actions. While doing so, it moves from one state to another. The response of an embedded system to its inputs depends on its current state. It is this behavior of an embedded system in response to inputs that is often modeled by a *finite state machine*. Relatively simpler systems, such as protocols, are modeled using finite state machines. More complex systems, such as the engine controller of an aircraft, are modeled using *statecharts* which can be considered as a generalization of finite state machines. In this section we focus on finite state machines; statecharts are described in a chapter devoted to statecharts in Volume 2. The next example illustrates a simple model using a finite state machine.

**EXAMPLE 3.1.** Consider a traditional table lamp that has a three-way rotary switch. When the switch is turned to one of its positions, the lamp is in the OFF state. Moving the switch clockwise to the next position moves the lamp to an ON-DIM state. Moving the switch further clockwise moves the lamp to the ON-BRIGHT state. Finally, moving the switch clockwise one more time brings the lamp back to the OFF state. The switch serves as the input device that controls the state of the lamp. The change of lamp state in response to the switch position is often illustrated using a *state* diagram as in Figure 3.2(a).

Our lamp has three states OFF, ON_DIM, and ON_BRIGHT, and one input. Note that the lamp switch has three distinct positions though from a lamp user's perspective the switch can only be turned clockwise to its next position. Thus "turning the switch one notch clockwise" is the only input. Suppose that the switch also can be moved counterclockwise. In this latter case, the lamp has two inputs one corresponding to the clockwise motion and the other to the counterclockwise motion. The number of distinct states of the lamp remains three but the state diagram is now as in Figure 3.2(b). ■

In the example above we have modeled the table lamp in terms of its states, inputs and transitions. In most practical embedded systems, the application of an input might cause the